

Inheritance and polymorphism

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

Object extensions

A `class` can be “extended”, adding new member data and functions to it

The “extended” `class` definition can make use of the existing one:

- the “new” `class` has all the member data and functions of the “old” one,
- the “new” class can (usually) be used anywhere the “old” one is required,
- this mechanism is called “inheritance”.

Example:

a 3D vector can be seen as a 2D vector plus a z component

Object behaviour

A `class` inheriting from another one can modify its member functions and define a specific behaviour

The “extended” `class` can (usually) be used where its “base” is required:

- the object can behave according to its actual type, and not the base type, e.g.:
 - triangles, squares, hexagons are all geometrical shapes,
 - they can be described by classes sharing a common interface,
 - functions to compute their areas and perimeters must be different;
- this functionality is called “polymorphism”.

Public inheritance

A class is said to “publicly inherit” from another one when it can be used anywhere in place of its base

```
class Vector3D: public Vector2D {
public:
    Vector3D(float x, float y, float z);
    ~Vector3D();
    float getZ() const;
    float mod() const;
private:
    float zv;
};
```

When a class “privately inherit” from another one it includes all its members and functions but it cannot be used in place of its base

Base initialization

The derived class (e.g. `Vector3D`) has an “hidden” member with base class type (e.g. `Vector2D`)

- Requests for members of the base class are automatically forwarded to it.
- The base object is created before the execution of the constructor of the derived object.
- An explicit initialization is required if the base class does not provide a default constructor.

```
Vector3D::Vector3D() :  
    zv(0.0) { // implicit default construction  
}  
Vector3D::Vector3D(float x, float y, float z) :  
    Vector2D(x, y), // explicit construction  
    zv(z) {  
}
```

Function redefinition

Base class functions can be re-defined by the derived class

- All the functions of the base class with the same name as any function in the derived class are hidden (independently on the parameters).
- The function of the base class is actually called if a derived object is accessed as a base object.

```
float Vector3D::mod() {
    return sqrt((getX()*getX())+
               (getY()*getY())+(zv*zv));
}

...
Vector3D u(3.4, 4.5, 7.2);
cout << u.mod() << endl; // Vector3D::mod()
Vector2D& v=u;
cout << v.mod() << endl; // Vector2D::mod()
```

Function resolution

The base `class` function can be called by using the scope resolution operator

```
float Vector3D::mod() {  
    float m2d = Vector2D::mod();  
    return sqrt((m2d*m2d)+(zv*zv));  
}
```

Name hiding

The declaration of a function in a derived `class` hides all the functions with the same name in the base `class`, also the ones with different parameters.

Access rights for derived classes

Derived class functions may have “special” rights in the access to base class members: `protected` specifier

- Members declared as `protected` can be accessed by the same class and the derived ones.
- Members declared as `protected` cannot be accessed by other classes and functions.

Non-public inheritance

A derived class may have a private or protected base

- The derived class may access public and protected members of the base.
- A derived object cannot be used in place of a private or protected base object.

Base member access specifier	Base class access specifier		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

Name “unhiding” and inheritance prevention

C++11 only

An hidden name can be “unhidden” with an `using` declaration.

```
class Vector3D: public Vector2D {
public:
    ...
    using Vector2D::transform;
    ...
};
```

C++11 only

A class cannot be used as base if declared `final`.

```
class Pippo final {
    ...
};
```

virtual functions

Functions of a base `class` can be declared to be “automatically” replaced by functions of the derived `class` even when called through the base `class` interface

Functions behaving in this way are called “virtual” functions.

```
class Shape {
public:
    ...
    virtual float perimeter() const;
    virtual float area() const;
    ...
};
```

Function override

C++11 only

A function in a derived `class` can be explicitly declared to `override` a `virtual` function and not to be a new function.

```
class Triangle: public Shape {  
    ...  
    float area() const override;  
    ...  
};
```

C++11 only

A function in a base `class` can prevent overriding.

```
class Shape {  
    ...  
    virtual void print() const final;  
    ...  
};
```

Function overriding, old style

```
class Triangle: public Shape {
public:
    ...
    virtual float area() const;
    ...
};
```

```
class Square: public Shape {
public:
    ...
    virtual float area() const;
    ...
};
```

- The “virtual” specification can be included in the function re-declaration, but this is not required.
- Re-declaring as “virtual” a function declared “non-virtual” in the base class is ineffective.

virtual functions call

Calls of a `virtual` function by pointers or references query a “virtual table”, and the function of the actual (derived) object is selected

```
Shape* t = new Triangle(...);
Shape* s = new Square(...);
cout << t->area() << endl;
cout << s->perimeter() << endl;
Shape& r = *s;
cout << r.area() << endl;
```

The “virtual table” is used in the call of `virtual` functions inside the base class functions, too.

```
void Shape::print() {
    cout << "perimeter: " << perimeter()
         << " and area: " << area() << endl;
}
```

Functions call by concrete objects

In the call of `virtual` functions by a concrete object the object function is (obviously) used.

```
Shape* s;  
...  
Shape& r = *s;  
Shape o = *s; // a copy of the object  
              // is created  
r.print(); // the actual object (pointed by s)  
           // function is called  
o.print(); // Shape::print() is called
```

- If a derived object is copied onto a base object, the “extensions” are lost (“slicing”).
- Special care is required when an object is passed to a function: calls by value could alter the object type and behaviour.

Destruction of derived classes

Derived objects can be deleted through pointers to base class objects

- Any resource allocated by the derived object must be released when the object is deleted, however this is done.
- The destructor should be declared `virtual`.

```
class Base {  
public:  
    Base();  
    virtual ~Base();  
    ...  
};
```

Function override summary

Function override for derived objects
accessed through base interface

Access mode	Function declaration	
	virtual	non-virtual
pointer	YES	NO
reference	YES	NO
copy	NO	NO

Construction and destruction sequence

Derived objects contain an instance of their base

- The base `class` constructor is run before the derived `class` constructor.
- The base `class` destructor is run after the derived `class` destructor.
- When the base `class` constructor is executed, the derived object has not yet been created.
- When the base `class` destructor is executed, the derived object has already been deleted.

The base `class` constructor and destructor cannot call the derived `class` functions (including overridden ones).

Pure virtual functions declaration

A class can contain functions to be necessarily (re)implemented in derived classes, called “pure virtual functions”

Member functions are declared being pure virtual by a trailing “ = 0 ”.

```
class Shape {  
public:  
    ...  
    virtual float perimeter() const = 0;  
    virtual float area() const = 0;  
    ...  
};
```

Pure virtual functions implementation

A pure virtual function can be implemented also in the base class, to be called with scope resolution.

```
class Base {
    ...
    virtual void f() = 0;
    ...
};
void Base::f() {
    ...
}
```

```
class Derived:
    public Base {
    ...
    void f() override;
    ...
};
void Derived::f() {
    ...
    Base::f();
    ...
}
```

Abstract base classes

Base classes with at least one pure virtual function are called “abstract base classes”

- No instance of an abstract base class can be created.
- Only instances of derived classes implementing all pure virtual functions can be created.
- A (direct or indirect) call to a pure virtual function in the base class constructor or destructor results in a fatal error.

Base to derived type casts

A pointer to a base `class` object can be “forced” to point to a derived `class` object

The operation returns a null pointer if the actual object type does not match the required one.

```
Shape* s;  
...  
Square* q = dynamic_cast<Square*>(s);  
if(q!=nullptr) cout << q->side() << endl;
```

A `dynamic_cast` of a reference throws an exception if the actual object type does not match the required one.

Multiple bases

A class can derive from multiple bases

```
class BaseA {  
    ...  
};  
class BaseB {  
    ...  
};  
class Derived: public BaseA, public BaseB {  
    ...  
};
```

- The derived class gets all the data and function members of all its bases.
- Any call to a function having the same name in more than one base is ambiguous.

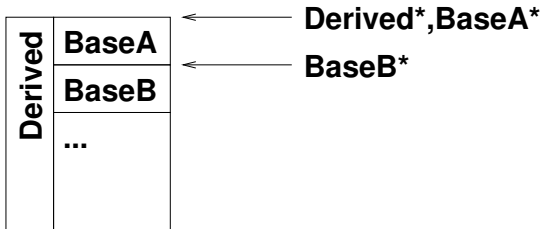
Function disambiguation

```
class BaseA {
    ...
    void f();
    ...
};
class BaseB {
    ...
    void f();
    ...
};
...
Derived* d;
...
d->f(); // ambiguous
d->BaseA::f();
...
```

Pointers and multiple bases

Pointers to objects derived from multiple bases can change when copied onto pointers to a base class

- The derived `class` has an “hidden” member for each base.
- They are stored in different memory locations.



Inheritance from derived classes

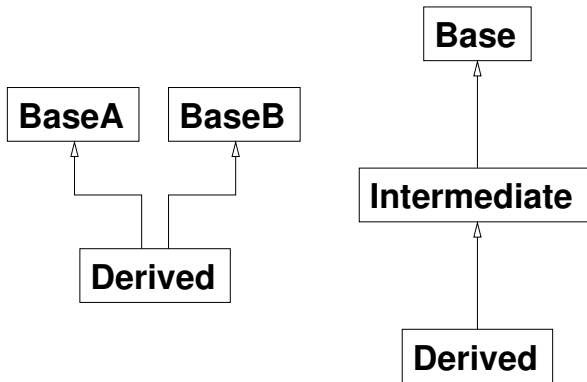
A class can derive from a class deriving in its turn from another one

```
class Base {  
    ...  
};  
class Intermediate: public Base {  
    ...  
};  
class Derived: public Intermediate {  
    ...  
};
```

The derived class gets the members of its “immediate” base, and the members of the “next” base according to their visibility.

UML graphs

Relations among `classes` can be drawn in “UML graphs”



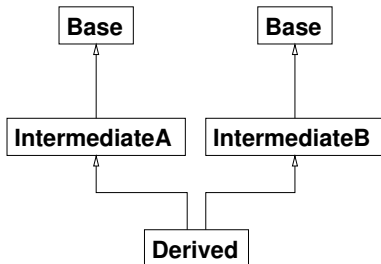
Inheritance is shown by an arrow pointing from the `derived class` to the `base class` .

Multiple inheritance with common bases

A class can derive from multiple bases in their turn inheriting from a common base

```
class Base {
    ...
};
class IntermediateA: public Base {
    ...
};
class IntermediateB: public Base {
    ...
};
class Derived: public IntermediateA,
               public IntermediateB {
    ...
};
```

Multiple inheritance disambiguation



- Each intermediate object contains a base object
- A call to a `Base` function from a `Derived` object can be ambiguous.

Such a structure is allowed, but
`Base` function calls from `Derived` need disambiguation

```
Derived* d;
...
d->IntermediateA::f();
d->IntermediateB::f();
```

virtual inheritance

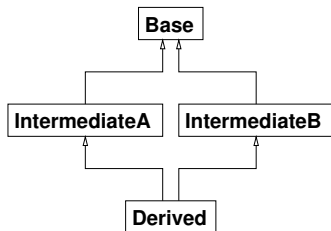
A “common” base can be declared to be shared by all its derived classes

- This can be achieved by including a `virtual` keyword before the base class name.
- The base class becomes a **direct base** for all derived classes in the inheritance chain.

```
class Base {
    ...
};
class IntermediateA: public virtual Base {
    ...
};
class IntermediateB: public virtual Base {
    ...
};
```

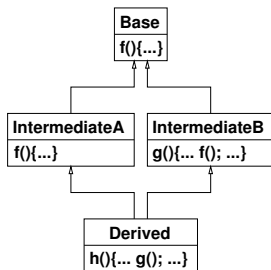
Diamond graphs

virtual inheritance allows to implement the so-called “diamond graph”



- Base can be created and initialized by any of IntermediateA, IntermediateB or Derived.
- When a Derived object is created:
 - Base is first created by Derived constructor,
 - IntermediateA and IntermediateB are then created,
 - the creation of Base in IntermediateA and IntermediateB constructors is neglected.

virtual inheritance and virtual functions



- Base can declare and implement a virtual function `f()` ,
- IntermediateA can override that function,
- a function `g()` of IntermediateB can call `f()` ,
- when `g()` is called:
 - for an IntermediateB object:
 - g() in its turn calls `Base::f()` ,
 - for a Derived object:
 - g() in its turn calls `IntermediateA::f()` .